属性 (Properties)

1.0 翻译：shinyzhu (https://github.com/shinyzhu) 校对：pp-prog (https://github.com/pp-prog) yangsiy (https://github.com/yangsiy)

2.0 翻译+校对：yangsiy (https://github.com/yangsiy)

2.1 翻译：buginux (https://github.com/buginux) 校对：shanks (http://codebuild.me)，2015-10-29

2.2 翻译：saitjr (https://github.com/saitjr)，2016-04-11，SketchK (https://github.com/SketchK) 2016-05-13

本页包含内容：

*属性*将值跟特定的类、结构或枚举关联。存储属性存储常量或变量作为实例的一部分，而计算属性计算（不是存储）一个值。计算属性可以用于类、结构体和枚举，存储属性只能用于类和结构体。

存储属性和计算属性通常与特定类型的实例关联。但是，属性也可以直接作用于类型本身，这种属性称为类型属性。

另外，还可以定义属性观察器来监控属性值的变化，以此来触发一个自定义的操作。属性观察器可以添加到自己定义的存储属性上，也可以添加到从父类继承的属性上。

存储属性

简单来说，一个存储属性就是存储在特定类或结构体实例里的一个常量或变量。存储属性可以是*变量存储属性*（用关键字 `var` 定义），也可以是*常量存储属性*（用关键字 `let` 定义）。

可以在定义存储属性的时候指定默认值，请参考默认构造器 (./14_Initialization.html#default_initializers)一节。也可以在构造过程中设置或修改存储属性的值，甚至修改常量存储属性的值，请参考构造过程中常量属性的修改 (./14_Initialization.html#assigning_constant_properties_during_initialization)一节。

下面的例子定义了一个名为 `FixedLengthRange` 的结构体，该结构体用于描述整数的范围，且这个范围值在被创建后不能被修改.

```
struct FixedLengthRange {
    var firstValue: Int
    let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length: 3)
// 该区间表示整数0，1，2
rangeOfThreeItems.firstValue = 6
// 该区间现在表示整数6，7，8
```

`FixedLengthRange` 的实例包含一个名为 `firstValue` 的变量存储属性和一个名为 `length` 的常量存储属性。在上面的例子中， `length` 在创建实例的时候被初始化，因为它是一个常量存储属性，所以之后无法修改它的值。

常量结构体的存储属性

如果创建了一个结构体的实例并将其赋值给一个常量，则无法修改该实例的任何属性，即使有属性被声明为变量也不行：

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 该区间表示整数0，1，2，3
rangeOfFourItems.firstValue = 6
// 尽管 firstValue 是个变量属性，这里还是会报错
```

因为 `rangeOfFourItems` 被声明成了常量（用 `let` 关键字），即使 `firstValue` 是一个变量属性，也无法再修改它了。

这种行为是由于结构体（struct）属于*值类型*。当值类型的实例被声明为常量的时候，它的所有属性也就成了常量。

属于*引用类型*的类（class）则不一样。把一个引用类型的实例赋给一个常量后，仍然可以修改该实例的变量属性。

延迟存储属性

延迟存储属性是指当第一次被调用的时候才会计算其初始值的属性。在属性声明前使用 `lazy` 来标示一个延迟存储属性。

> 注意
> 必须将延迟存储属性声明成变量（使用 `var` 关键字），因为属性的初始值可能在实例构造完成之后才会得到。而常量属性在构造过程完成之前必须要有初始值，因此无法声明成延迟属性。

延迟属性很有用，当属性的值依赖于在实例的构造过程结束后才会知道影响值的外部因素时，或者当获得属性的初始值需要复杂或大量计算时，可以只在需要的时候计算它。

下面的例子使用了延迟存储属性来避免复杂类中不必要的初始化。例子中定义了 `DataImporter` 和 `DataManager` 两个类，下面是部分代码：

```
class DataImporter {
    /*
    DataImporter 是一个负责将外部文件中的数据导入的类。
    这个类的初始化会消耗不少时间。
    */
    var fileName = "data.txt"
    // 这里会提供数据导入功能
}

class DataManager {
    lazy var importer = DataImporter()
    var data = [String]()
    // 这里会提供数据管理功能
}

let manager = DataManager()
manager.data.append("Some data")
manager.data.append("Some more data")
// DataImporter 实例的 importer 属性还没有被创建
```

`DataManager` 类包含一个名为 `data` 的存储属性，初始值是一个空的字符串（ `String` ）数组。这里没有给出全部代码，只需知道 `DataManager` 类的目的是管理和提供对这个字符串数组的访问即可。

`DataManager` 的一个功能是从文件导入数据。该功能由 `DataImporter` 类提供， `DataImporter` 完成初始化需要消耗不少时间：因为它的实例在初始化时可能要打开文件，还要读取文件内容到内存。

`DataManager` 管理数据时也可能不从文件中导入数据。所以当 `DataManager` 的实例被创建时，没必要创建一个 `DataImporter` 的实例，更明智的做法是第一次用到 `DataImporter` 的时候才去创建它。

由于使用了 `lazy` ， `importer` 属性只有在第一次被访问的时候才被创建。比如访问它的属性 `fileName` 时：

```
print(manager.importer.fileName)
// DataImporter 实例的 importer 属性现在被创建了
// 输出 "data.txt”
```

> 注意
> 如果一个被标记为 `lazy` 的属性在没有初始化时就同时被多个线程访问，则无法保证该属性只会被初始化一次。

存储属性和实例变量

如果您有过 Objective-C 经验，应该知道 Objective-C 为类实例存储值和引用提供两种方法。除了属性之外，还可以使用实例变量作为属性值的后端存储。

Swift 编程语言中把这些理论统一用属性来实现。Swift 中的属性没有对应的实例变量，属性的后端存储也无法直接访问。这就避免了不同场景下访问方式的困扰，同时也将属性的定义简化成一个语句。属性的全部信息——包括命名、类型和内存管理特征——都在唯一一个地方（类型定义中）定义。

计算属性

除存储属性外，类、结构体和枚举可以定义*计算属性*。计算属性不直接存储值，而是提供一个 getter 和一个可选的 setter，来间接获取和设置其他属性或变量的值。

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set(newCenter) {
            origin.x = newCenter.x - (size.width / 2)
            origin.y = newCenter.y - (size.height / 2)
        }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
print("square.origin is now at (\(square.origin.x), \(square.origin.y))")
// 输出 "square.origin is now at (10.0, 10.0)”
```

这个例子定义了 3 个结构体来描述几何形状：

- `Point` 封装了一个 `(x, y)` 的坐标
- `Size` 封装了一个 `width` 和一个 `height`
- `Rect` 表示一个有原点和尺寸的矩形

`Rect` 也提供了一个名为 `center` 的计算属性。一个矩形的中心点可以从原点（ `origin` ）和大小（ `size` ）算出，所以不需要将它以显式声明的 `Point` 来保存。`Rect` 的计算属性 `center` 提供了自定义的 getter 和 setter 来获取和设置矩形的中心点，就像它有一个存储属性一样。

上述例子中创建了一个名为 `square` 的 `Rect` 实例，初始值原点是 `(0, 0)`，宽度高度都是 `10`。如下图中蓝色正方形所示。

`square` 的 `center` 属性可以通过点运算符（ `square.center` ）来访问，这会调用该属性的 getter 来获取它的值。跟直接返回已经存在的值不同，getter 实际上通过计算然后返回一个新的 `Point` 来表示 `square` 的中心点。如代码所示，它正确返回了中心点 `(5, 5)`。

`center` 属性之后被设置了一个新的值 `(15, 15)`，表示向右上方移动正方形到如下图橙色正方形所示的位置。设置属性 `center` 的值会调用它的 setter 来修改属性 `origin` 的 `x` 和 `y` 的值，从而实现移动正方形到新的位置。

便捷 setter 声明

如果计算属性的 setter 没有定义表示新值的参数名，则可以使用默认名称 `newValue`。下面是使用了便捷 setter 声明的 `Rect` 结构体代码：

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
        get {
            let centerX = origin.x + (size.width / 2)
            let centerY = origin.y + (size.height / 2)
            return Point(x: centerX, y: centerY)
        }
        set {
            origin.x = newValue.x - (size.width / 2)
            origin.y = newValue.y - (size.height / 2)
        }
    }
}
```

只读计算属性

只有 getter 没有 setter 的计算属性就是*只读计算属性*。只读计算属性总是返回一个值，可以通过点运算符访问，但不能设置新的值。

> 注意
> 必须使用 `var` 关键字定义计算属性，包括只读计算属性，因为它们的值不是固定的。 `let` 关键字只用来声明常量属性，表示初始化后再也无法修改的值。

只读计算属性的声明可以去掉 `get` 关键字和花括号：

```
struct Cuboid {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
        return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
print("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// 输出 "the volume of fourByFiveByTwo is 40.0"
```

这个例子定义了一个名为 `Cuboid` 的结构体，表示三维空间的立方体，包含 `width` 、 `height` 和 `depth` 属性。结构体还有一个名为 `volume` 的只读计算属性用来返回立方体的体积。为 `volume` 提供 setter 毫无意义，因为无法确定如何修改 `width` 、 `height` 和 `depth` 三者的值来匹配新的 `volume` 。然而， `Cuboid` 提供一个只读计算属性来让外部用户直接获取体积是很有用的。

属性观察器

*属性观察器*监控和响应属性值的变化，每次属性被设置值的时候都会调用属性观察器，即使新值和当前值相同的时候也不例外。

可以为除了延迟存储属性之外的其他存储属性添加属性观察器，也可以通过重写属性的方式为继承的属性（包括存储属性和计算属性）添加属性观察器。你不必为非重写的计算属性添加属性观察器，因为可以通过它的 setter 直接监控和响应值的变化。 属性重写请参考重写 (./13_Inheritance.html#overriding)。

可以为属性添加如下的一个或全部观察器：

- `willSet` 在新的值被设置之前调用
- `didSet` 在新的值被设置之后立即调用

`willSet` 观察器会将新的属性值作为常量参数传入，在 `willSet` 的实现代码中可以为这个参数指定一个名称，如果不指定则参数仍然可用，这时使用默认名称 `newValue` 表示。

同样， `didSet` 观察器会将旧的属性值作为参数传入，可以为该参数命名或者使用默认参数名 `oldValue` 。如果在 `didSet` 方法中再次对该属性赋值，那么新值会覆盖旧的值。

> 注意
> 父类的属性在子类的构造器中被赋值时，它在父类中的 `willSet` 和 `didSet` 观察器会被调用，随后才会调用子类的观察器。在父类初始化方法调用之前，子类给属性赋值时，观察器不会被调用。 有关构造器代理的更多信息，请参考值类型的构造器代理 (./14_Initialization.html#initializer_delegation_for_value_types)和类的构造器代理规则 (./14_Initialization.html#initializer_delegation_for_class_types)。

下面是一个 `willSet` 和 `didSet` 实际运用的例子，其中定义了一个名为 `StepCounter` 的类，用来统计一个人步行时的总步数。这个类可以跟计步器或其他日常锻炼的统计装置的输入数据配合使用。

```
class StepCounter {
    var totalSteps: Int = 0 {
        willSet(newTotalSteps) {
            print("About to set totalSteps to \(newTotalSteps)")
        }
        didSet {
            if totalSteps > oldValue  {
                print("Added \(totalSteps - oldValue) steps")
            }
        }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// Added 536 steps
```

`StepCounter` 类定义了一个 `Int` 类型的属性 `totalSteps`，它是一个存储属性，包含 `willSet` 和 `didSet` 观察器。

当 `totalSteps` 被设置新值的时候，它的 `willSet` 和 `didSet` 观察器都会被调用，即使新值和当前值完全相同时也会被调用。

例子中的 `willSet` 观察器将表示新值的参数自定义为 `newTotalSteps`，这个观察器只是简单的将新的值输出。

`didSet` 观察器在 `totalSteps` 的值改变后被调用，它把新值和旧值进行对比，如果总步数增加了，就输出一个消息表示增加了多少步。`didSet` 没有为旧值提供自定义名称，所以默认值 `oldValue` 表示旧值的参数名。

> 注意
>
> 如果将属性通过 in-out 方式传入函数，`willSet` 和 `didSet` 也会调用。这是因为 in-out 参数采用了拷入拷出模式：即在函数内部使用的是参数的 copy，函数结束后，又对参数重新赋值。关于 in-out 参数详细的介绍，请参考输入输出参数 (../chapter3/05_Declarations.html#in-out_parameters)

全局变量和局部变量

计算属性和属性观察器所描述的功能也可以用于*全局变量*和*局部变量*。全局变量是在函数、方法、闭包或任何类型之外定义的变量。局部变量是在函数、方法或闭包内部定义的变量。

前面章节提到的全局或局部变量都属于存储型变量，跟存储属性类似，它为特定类型的值提供存储空间，并允许读取和写入。

另外，在全局或局部范围都可以定义计算型变量和为存储型变量定义观察器。计算型变量跟计算属性一样，返回一个计算结果而不是存储值，声明格式也完全一样。

> 注意
> 全局的常量或变量都是延迟计算的，跟延迟存储属性相似，不同的地方在于，全局的常量或变量不需要标记 `lazy` 修饰符。
> 局部范围的常量或变量从不延迟计算。

类型属性

实例属性属于一个特定类型的实例，每创建一个实例，实例都拥有属于自己的一套属性值，实例之间的属性相互独立。

也可以为类型本身定义属性，无论创建了多少个该类型的实例，这些属性都只有唯一一份。这种属性就是*类型属性*。

类型属性用于定义某个类型所有实例共享的数据，比如所有实例都能用的一个常量（就像 C 语言中的静态常量），或者所有实例都能访问的一个变量（就像 C 语言中的静态变量）。

存储型类型属性可以是变量或常量，计算型类型属性跟实例的计算型属性一样只能定义成变量属性。

> 注意
>
> 跟实例的存储型属性不同，必须给存储型类型属性指定默认值，因为类型本身没有构造器，也就无法在初始化过程中使用构造器给类型属性赋值。
>
> 存储型类型属性是延迟初始化的，它们只有在第一次被访问的时候才会被初始化。即使它们被多个线程同时访问，系统也保证只会对其进行一次初始化，并且不需要对其使用 `lazy` 修饰符。

类型属性语法

在 C 或 Objective-C 中，与某个类型关联的静态常量和静态变量，是作为全局（*global*）静态变量定义的。但是在 Swift 中，类型属性是作为类型定义的一部分写在类型最外层的花括号内，因此它的作用范围也就在类型支持的范围内。

使用关键字 `static` 来定义类型属性。在为类定义计算型类型属性时，可以改用关键字 `class` 来支持子类对父类的实现进行重写。下面的例子演示了存储型和计算型类型属性的语法：

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
}
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 6
    }
}
class SomeClass {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 27
    }
    class var overrideableComputedTypeProperty: Int {
        return 107
    }
}
```

> 注意
>
> 例子中的计算型类型属性是只读的，但也可以定义可读可写的计算型类型属性，跟计算型实例属性的语法相同。

获取和设置类型属性的值

跟实例属性一样，类型属性也是通过点运算符来访问。但是，类型属性是通过类型本身来访问，而不是通过实例。比如：

```
print(SomeStructure.storedTypeProperty)
// 输出 "Some value."
SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// 输出 "Another value.”
print(SomeEnumeration.computedTypeProperty)
// 输出 "6"
print(SomeClass.computedTypeProperty)
// 输出 "27"
```

下面的例子定义了一个结构体，使用两个存储型类型属性来表示两个声道的音量，每个声道具有 `0` 到 `10` 之间的整数音量。

下图展示了如何把两个声道结合来模拟立体声的音量。当声道的音量是 `0`，没有一个灯会亮；当声道的音量是 `10`，所有灯点亮。本图中，左声道的音量是 `9`，右声道的音量是 `7`：

上面所描述的声道模型使用 `AudioChannel` 结构体的实例来表示：

```
struct AudioChannel {
    static let thresholdLevel = 10
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
        didSet {
            if currentLevel > AudioChannel.thresholdLevel {
                // 将当前音量限制在阀值之内
                currentLevel = AudioChannel.thresholdLevel
            }
            if currentLevel > AudioChannel.maxInputLevelForAllChannels {
                // 存储当前音量作为新的最大输入音量
                AudioChannel.maxInputLevelForAllChannels = currentLevel
            }
        }
    }
}
```

结构 `AudioChannel` 定义了 2 个存储型类型属性来实现上述功能。第一个是 `thresholdLevel`，表示音量的最大上限阈值，它是一个值为 `10` 的常量，对所有实例都可见，如果音量高于 `10`，则取最大上限值 `10`（见后面描述）。

第二个类型属性是变量存储型属性 `maxInputLevelForAllChannels`，它用来表示所有 `AudioChannel` 实例的最大音量，初始值是 `0`。

`AudioChannel` 也定义了一个名为 `currentLevel` 的存储型实例属性，表示当前声道现在的音量，取值为 `0` 到 `10`。

属性 `currentLevel` 包含 `didSet` 属性观察器来检查每次设置后的属性值，它做如下两个检查：

- 如果 `currentLevel` 的新值大于允许的阈值 `thresholdLevel`，属性观察器将 `currentLevel` 的值限定为阈值 `thresholdLevel`。
- 如果修正后的 `currentLevel` 值大于静态类型属性 `maxInputLevelForAllChannels` 的值，属性观察器就将新值保存在 `maxInputLevelForAllChannels` 中。

> 注意
> 在第一个检查过程中，`didSet` 属性观察器将 `currentLevel` 设置成了不同的值，但这不会造成属性观察器被再次调用。

可以使用结构体 `AudioChannel` 创建两个声道 `leftChannel` 和 `rightChannel`，用以表示立体声系统的音量：

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

如果将左声道的 `currentLevel` 设置成 `7`，类型属性 `maxInputLevelForAllChannels` 也会更新成 `7`：

```
leftChannel.currentLevel = 7
print(leftChannel.currentLevel)
// 输出 "7"
print(AudioChannel.maxInputLevelForAllChannels)
// 输出 "7"
```

如果试图将右声道的 `currentLevel` 设置成 `11`，它会被修正到最大值 `10`，同时 `maxInputLevelForAllChannels` 的值也会更新到 `10`：

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// 输出 "10"
print(AudioChannel.maxInputLevelForAllChannels)
// 输出 "10"
```

上一篇: 类和结构体 (/project/swift/chapter2/09_Classes_and_Structures.html)

下一篇: 方法 (/project/swift/chapter2/11_Methods.html)

被顶起来的评论

dddd

其实我比较想问为什么didset里面再设置值不会死循环

2015年12月14日 回复 顶(2) 转发

从今以后

回复 沉沦2013: 并没有这种要求，只要保证构造过程结束时所有存储型属性都有初始值，或者是延迟加载的即可~

2015年11月16日 回复 顶(1) 转发

可爱未遂的小思思 (http://weibo.com/veightz)

回复 馬资懿: class BaseClass {
static var computedTypeProperty: Int {
return 1
}
class var overrideableComputedTypeProperty: Int {
return 2
}
}

class SomeClass: BaseClass {
override class var overrideableComputedTypeProperty: Int {
return 3
}
}
Class 的类型变量中，使用 class 而不是 static 标记的类型变量可以被 override，一开始我也理解错意思了。

2015年11月4日 回复 顶(1) 转发

29条评论 1条新浪微博 最新 最早 最热

寒桥 (http://t.qq.com/taotao88662009)

let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// 该区间表示整数0，1，2，3
rangeOfFourItems.firstValue = 6
// 尽管 firstValue 是个变量属性，这里还是会报错

这个在我这边没有报错似乎是支持这种类型 不知道为什么 其他人在playground上有没有遇到这种情况呢?

6月4日 回复 顶 转发

xiewh

属性观察器中这句话：“你不必为非重写的计算属性添加属性观察器”，改为“不必为没有重写的计算属性添加属性观察器”是不是更好?

5月13日 回复 顶 转发

陈兴Startry (http://weibo.com/chenxingstartry)

回复 从今以后: 其实是官方文档就是这样写的, 所以比较容易误导~
https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID254 (https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID254)

2月29日 回复 顶 转发

永_远_有_多_远 (http://weibo.com/2651440571)

已阅，计算属性、存储属性、类型属性不算是比较新的概念，其他语言也有这些东东，可能叫法不一样，刨除里面掺杂的继承的东西，还是很好理解的

2月25日 回复 顶 转发

tpphha

回复 可爱未遂的小思思: 汗, 我也理解错了

2月18日 回复 顶 转发

悉毅 (http://weibo.com/1403384842)

求教这玩意的应用场景，用例是什么？以前Obj-C上有没有类似的功能?

(http://weibo.com/1403384842)1月25日 回复 顶 转发

hosea

回复 dddd: 不会的。只会执行一次~

2015年12月30日 回复 顶 转发

Lay

回复 dddd: didset 内和是例外处理应该是不一样的吧 didset内不会将值作为实例的属性引用处理 而是作为newValue处理

2015年12月21日 回复 顶 转发

太极正能量o_O (http://weibo.com/1671908302)

回复 沉沦2013: 应该为nil

(http://weibo.com/1671908302)2015年12月17日 回复 顶 转发

dddd

其实我比较想问为什么didset里面再设置值不会死循环

2015年12月14日 回复 顶(2) 转发

dddd

回复 caoping: 重写的不是存储属性,而是计算属性,计算属性就可以理解为类方法,所以子类重写父类的类方法是很正常的.....

2015年12月14日 回复 顶 转发

caoping (http://weibo.com/caoping)

回复 笑你妹: 重写父类static 属性是个什么鬼?

(http://weibo.com/caoping)2015年12月8日 回复 顶 转发

笑你妹 (http://t.qq.com/ytx2577)

class 重写父类的static, 是什么情况?

(http://t.qq.com/ytx2577)2015年11月26日 回复 顶 转发

从今以后

回复 陈兴Startry: 是有点，我还以为校对错了

2015年11月16日 回复 顶 转发

从今以后

回复 沉沦2013: 并没有这种要求，只要保证构造过程结束时所有存储型属性都有初始值，或者是延迟加载的即可~

2015年11月16日 回复 顶(1) 转发

Nero

回复 沉沦2013: class也是可以没有初始值的，但必须给定定义init方法，在构造函数中实现初始化，而struct允许这么写的原因是struct不写构造函数，会默认给构造函数

2015年11月13日 回复 顶 转发

恩赐

回复 沉沦2013: 什么时候要求所有的变量和常量在创建时必须有初始值了？。。。。。。

2015年11月12日 回复 顶 转发

沉沦2013

有问题请教一下，swift中要求所有常量或者变量在创建的时候必须有初始值，
class有初始值
class DataManager {
lazy var importer = DataImporter()
var data = [String]()
}
为什么在上面的
struct FixedLengthRange {
var firstValue: Int
let length: Int
}中没有初始值也是允许的?

2015年11月8日 回复 顶 转发

馬资懿 (http://weibo.com/emosad)

回复 可爱未遂的小思思: 明白了，可是后面章节讲到final的时候例举了 final class func 的例子 我就在想，既然如此直接写static func 不就好了嘛。但是在想如果父类需要overrideable，子类的子类不需要就可以用final，子类的同级子类则可以override

(http://weibo.com/emosad)2015年11月4日 回复 顶 转发

可爱未遂的小思思 (http://weibo.com/veightz)

回复 可爱未遂的小思思: 这缩进。。。

(http://weibo.com/veightz)2015年11月4日 回复 顶 转发